



Cognitier SimpleIPC

Multithreaded MS Excel Automation With Siebel Sample

Version 1.0.0.1

June 02, 2009

Copyright © 2009 Cognitier, Inc. All rights reserved.

Cognitier and the Cognitier logo are trademarks of Cognitier, Inc. All other company and product names referred to may be trademarks of their respective owners. This documentation is copyrighted material and is intended exclusively for use by licensed users of Cognitier software. The information in this document is subject to change without notice.

Multithreaded MS Excel Automation With Siebel Sample

Implementing multithreaded behavior in Microsoft Office applications can be challenging. Establishing background tasks via VBA timers can be problematic because the background task is consuming the MS Office application's resources. The MS Office application itself may become unresponsive. Memory consumption may increase, and if there is a problem in the VBA code, it may lead to a crash of the MS Office application.

In this example, the goal is to populate an MS Excel spreadsheet from a Siebel data source. The data in the spreadsheet will be refreshed on a scheduled basis. There will be a timer established in VBA. However, the timer will simply create a "refresh" request in SimpleIPC and return. This is done to minimize the time during which VBA code is running, with the objective of making the Excel application itself more responsive to the user. Within SimpleIPC, our custom routine will query Siebel and prepare a data set. It will then compare the data retrieved from Siebel against the data existing in the spreadsheet. It will update the spreadsheet cells where it finds a difference – so in most cases, it will not update anything. We want to minimize the probability of the routine and the interactive user updating the same cell at the same time. The queries against Siebel may take minutes to complete. We could run similar queries directly from VBA code, but we do not want to "tie up" the MS Excel application for this long while the queries are running – especially since it is likely that nothing has changed since the last query.

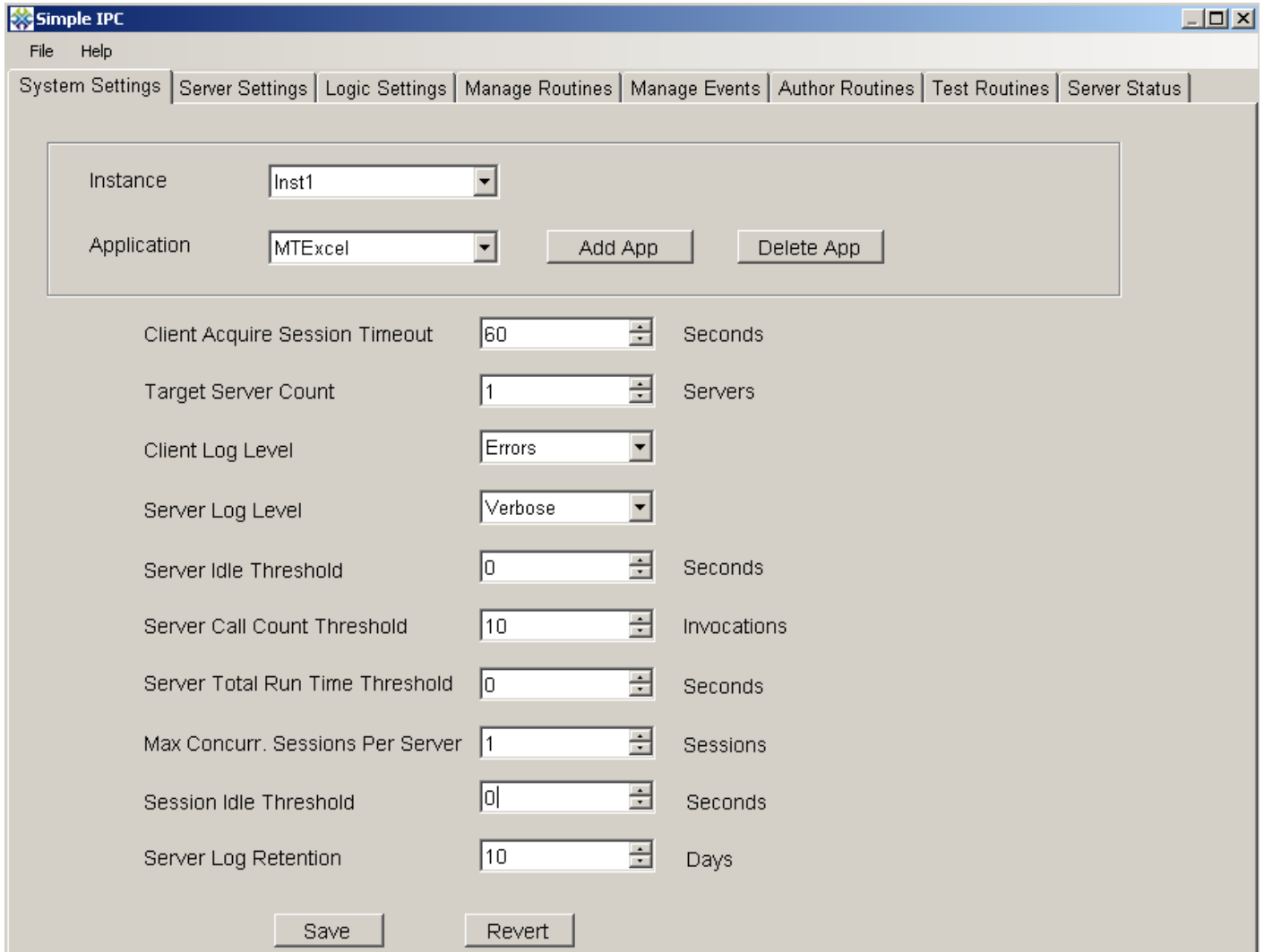
The general technical approach is that VBA code in the MS Excel application will run on a timer and create a request to refresh the spreadsheet contents. This is done via a SimpleIPC routine invocation. The arguments are the name of the MS Excel file and the set of ROW Ids (if any) already existing in the spreadsheet. The VBA code makes the request and returns immediately. The SimpleIPC routine processes requests in first-in-first-out order. When the Siebel data has been assembled, the routine will do the comparisons and update the spreadsheet as needed. When the MS Excel file is closed, a call will be made via VBA to discontinue any updates from SimpleIPC.

This example presents a simple case where there is one MS Excel file and one data source. This could be expanded such that the SimpleIPC server refreshes data for multiple spreadsheets, and polls multiple data sources for the data. This example does not illustrate the presentation of a graph or chart in MS Excel from the data being refreshed. This would likely be a requirement in a production application, but it is more of a general MS Excel automation exercise.

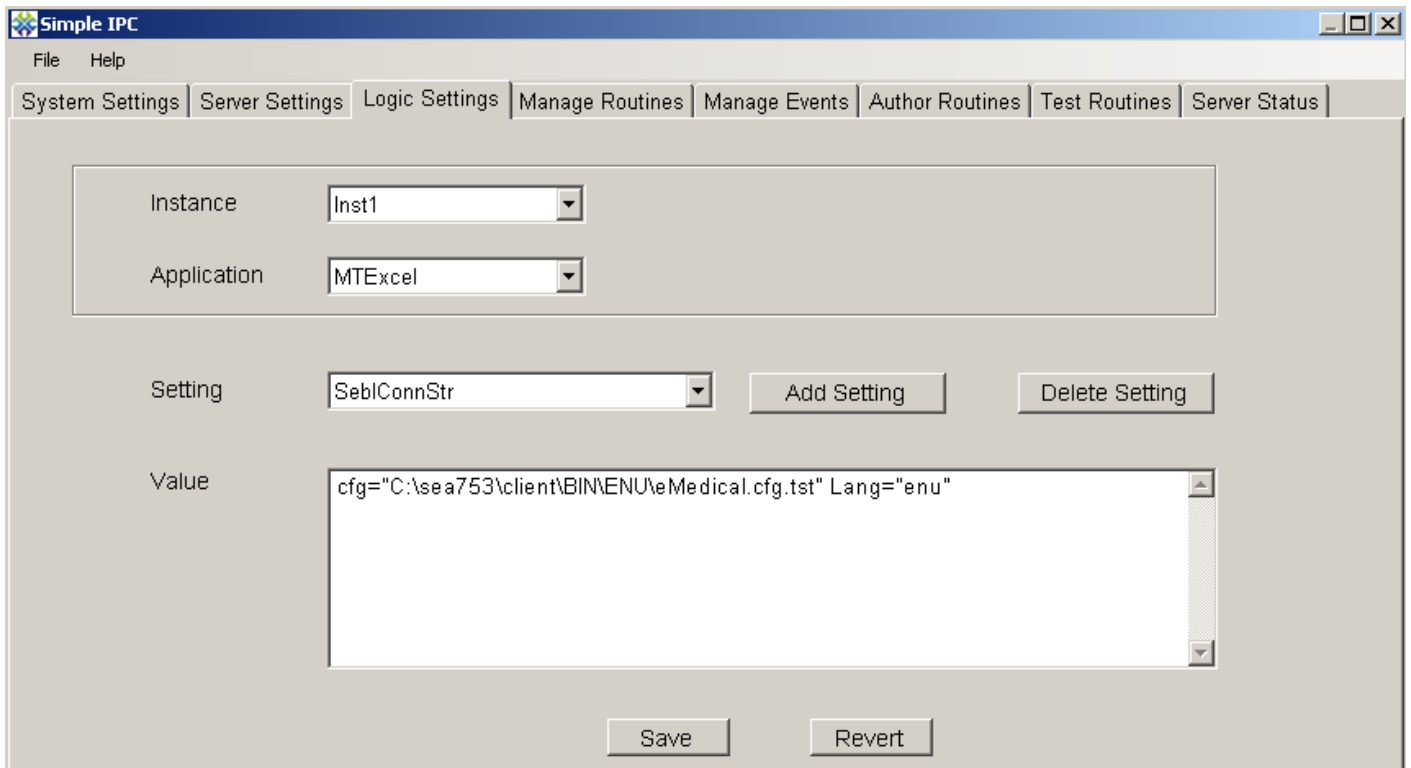
Important Note: You can configure SimpleIPC to run under a fixed Windows identity. However, in order for SimpleIPC to interact with the same spreadsheet on which the user is working, the identity must be set to "Launching User". Within VBA code, you can make a SimpleIPC call and obtain return values and update cells from those return values (i.e. not using automation). If you took this approach, then the IPC server could run under a different fixed identity. If you had the need to use SimpleIPC with automation for some data, and use a different fixed identity to retrieve other data, then you would need to set one SimpleIPC instance to run as "Launching User", and the other to run under the set identity via DCOMCONFIG.

The sample code omits many error-checking operations in order to make the code more readable.

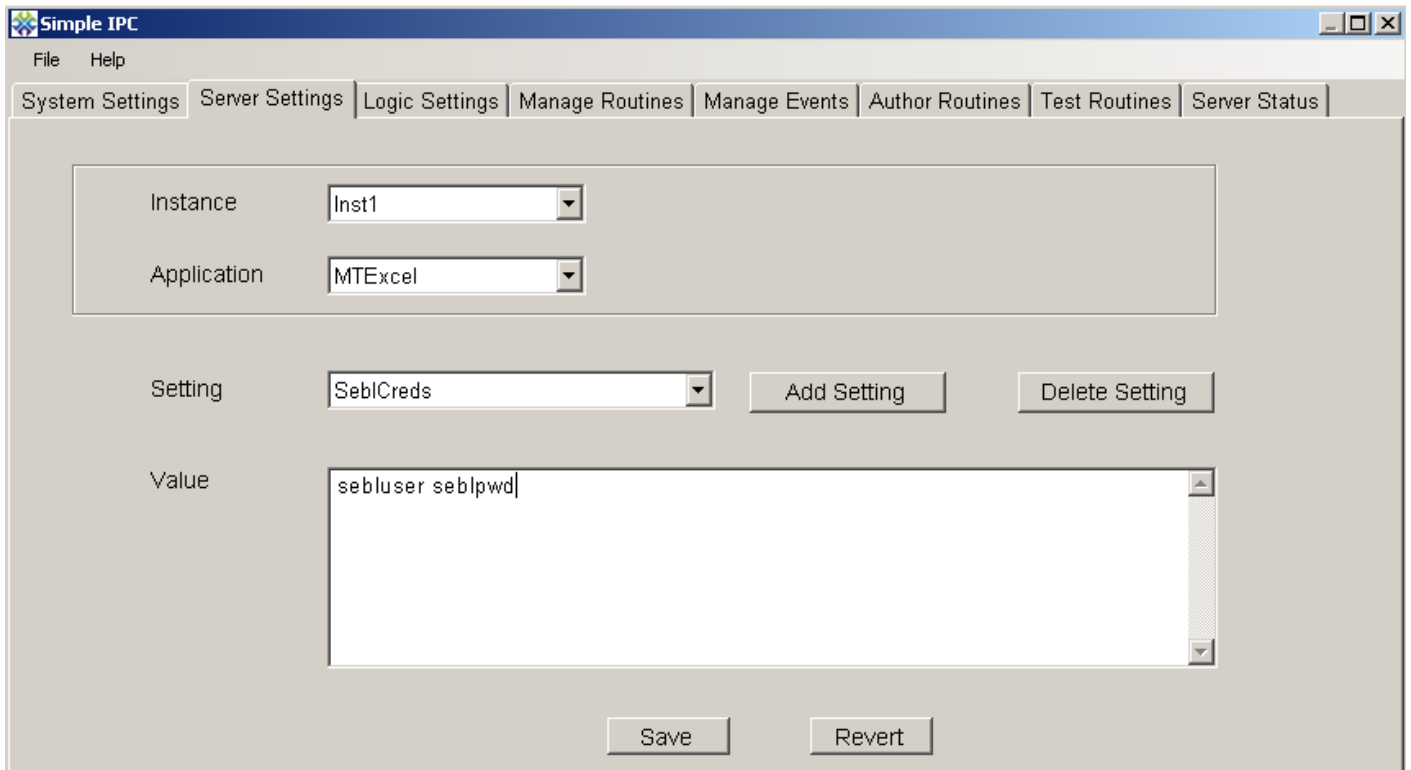
Step 1: Go to the System Settings tab in the CTConsole and create a new application in SimpleIPC for this exercise. Call it MTEExcel. Designate that there may be one server for this Instance/Application combination, and one session per server. Set the server idle threshold to zero (infinite), the server total run time threshold to zero (infinite), and the session idle threshold to zero (infinite). Set the server call count threshold to ten calls, and set the server log level to verbose.



Step 2: Create a Logic Setting for this new application with the connection string to use when connecting to Siebel via the COM Data Control in thick client mode. Go to the Logic Settings tab and toggle the Instance selection in order to refresh the list of Applications. Select the MTEExcel application and enter the new Logic Setting. Call the Logic Setting SeblConnStr, enter a value appropriate for your environment, and save the value.



Step 3: Create a Server Setting for this new application with the credentials to use when connecting to Siebel. Go to the Server Settings tab and toggle the Instance selection in order to refresh the list of Applications. Select the MTEExcel application and enter the new Server Setting. We could create separate settings for the username and password, but in this case we will combine both values into one setting delimited by the space character. Call the Server Setting SeblCreds, enter a value appropriate for your environment, and save the value.



Step 4: Go to the Author Routines tab and create the routine that will execute when an IPC server starts. Create a new routine called MTEExcelStart using the BasicServerOp1 template. Enter the following code to start a worker thread which connects to Siebel and then processes incoming client requests using Excel automation.

```
import System;
import System.Collections;
import System.Reflection;

[assembly: AssemblyVersion("1.0.0.0")]

public class MTEExcelStart implements CTSHost.IBasicServerOp1
{
    static var mServerContextAccessor: CTSHost.ServerContextAccessor;
    public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor) : System.Boolean
    {
        MTEExcelStart .mServerContextAccessor = oServerContextAccessor;
        var oLogUtil: CTCommon.LogUtil;
        var bRet: System.Boolean;
```

```

oLogUtil = CTCommon.LogUtil.Instance;
bRet = false;
try
{
    //Start a worker thread to keep running through the server's lifespan
    var oWorkerThread: System.Threading.Thread = new
System.Threading.Thread(System.Threading.ThreadStart(ProcessInboundRequests));
    oWorkerThread.SetApartmentState(System.Threading.ApartmentState.STA);
    oWorkerThread.Start();

    bRet = true;
}
catch (e)
{
    //Write errors to the server log file
    oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
}
return bRet;
} //end function

static function ProcessInboundRequests()
{
    var oLogUtil: CTCommon.LogUtil;
    oLogUtil = CTCommon.LogUtil.Instance;
    try
    {
        var htServerContextItems: Hashtable;
        var bStopPinging: Boolean = false;
        var oWorkBook = null;
        var oSheet = null;
        var iColCount: int = 0;
        var iRowCount: int = 0;
        var bWorksheetEmpty: Boolean = false;
        var sAcctRowID: String = String.Empty;
        //Cast the argument object back to the ServerContextAccessor

```

```

    var oServerContextAccessor: CTSHost.ServerContextAccessor =
MTEXcelStart.mServerContextAccessor;

    //Put a boolean variable in the context to tell us when to end thread
execution
    var bServerShuttingDown: System.Boolean = false;
    oServerContextAccessor.SetServerContextItem("ServerShuttingDown",
bServerShuttingDown);
    //Instantiate the COM Data Control
    var oSiebelApp = new
ActiveXObject("SiebelDataControl.SiebelDataControl.1");
    //Get the Logic Setting value with the connection string
    var sConnStr: String =
oServerContextAccessor.GetLogicSetting("SeblConnStr");
    //Get the Server Setting value with the concatenated username and password
    var sCreds: System.String =
oServerContextAccessor.GetServerSetting("SeblCreds");
    //Parse the concatenated credentials
    var sCredArray: String[] = sCreds.Split(" ");
    //Make the connection to Siebel
    oSiebelApp.Login(sConnStr, sCredArray[0], sCredArray[1]);
    //Get a reference to the Account Business Object
    var oAccountBO = oSiebelApp.GetBusObject("Account");
    //Get a reference to the Account Business Component
    var oAccountBC = oAccountBO.GetBusComp("Account");
    oAccountBC.ActivateField("Name");
    oAccountBC.ActivateField("City");
    oAccountBC.ActivateField("State");
    oAccountBC.ActivateField("Main Phone Number");

    //Get a reference to the Order Entry Business Object
    var oOrderBO = oSiebelApp.GetBusObject("Order Entry");
    //Get a reference to the Order Entry - Orders Business Component
    var oOrderBC = oOrderBO.GetBusComp("Order Entry - Orders");
    oOrderBC.ActivateField("Order Number");
    oOrderBC.ActivateField("Status");
    oOrderBC.ActivateField("Account Id");

```

```

//Get a reference to the Service Request Business Object
var oSRBO = oSiebelApp.GetBusObject("Service Request");
//Get a reference to the Service Request Business Component
var oSRBC = oSRBO.GetBusComp("Service Request");
oSRBC.ActivateField("SR Number");
oSRBC.ActivateField("Status");
oSRBC.ActivateField("Account Id");

//Prepare variables for handling requests
var sRequestID: String = String.Empty;
var oRequestPayload: ArrayList = new ArrayList();
var iRet: int = 0;
var iRecordCount: int = 0;
var sRequestingFile: String = String.Empty;
var oAcctRowIDs: ArrayList = new ArrayList();
var oAcctFields: ArrayList = new ArrayList();
var oAcctRecords: Hashtable = new Hashtable();

while (bServerShuttingDown== false)
{
    //Loop while the server doesn't appear to be shutting down
    bStopPinging = false;
    htServerContextItems =
oServerContextAccessor.GetServerContextItems();
    bServerShuttingDown =
System.Boolean(htServerContextItems["ServerShuttingDown"]);
    //Look for new requests queued in the server context
    sRequestID = String.Empty;
    for (var sKeyName in htServerContextItems.Keys)
    {
        if (sKeyName.StartsWith("REQ::"))
        {
            sRequestID = sKeyName;
            break;
        }
    }
}

```

```

        if (sRequestID.Length > 0)
        {
            //Get the request details from the server context item and then take
the item out of the context
            //The request details is an arraylist - the first element is the
file name for the spreadsheet
            //The second element is an arraylist with the Account ROW IDs
oRequestPayload = ArrayList(htServerContextItems[sRequestID]);
oServerContextAccessor.RemoveServerContextItem(sRequestID);
sRequestingFile = String(oRequestPayload(0));
oAcctRowIDs = ArrayList(oRequestPayload(1));
oAcctFields.Clear();
oAcctRecords = new Hashtable();
            //If the arraylist of account row ids was empty, go get the 10 most
recently-created accounts
            if (oAcctRowIDs.Count == 0)
            {
                oAccountBC.ClearToQuery();
                oAccountBC.SetViewMode(3);
                oAccountBC.SetSearchSpec("State", "TN");
                oAccountBC.SetSortSpec("Created(DSCENDING)");
                iRet = oAccountBC.ExecuteQuery(1);
                iRet = oAccountBC.FirstRecord();
                iRecordCount = 0;
                while (iRet != 0 && iRecordCount < 10)
                {
                    oAcctRowIDs.Add(oAccountBC.GetFieldValue("Id"));
                    iRet = oAccountBC.NextRecord();
                    iRecordCount++;
                }
            }
            //For each Account, query for certain attributes
            iRecordCount = 0;
            while (iRecordCount < oAcctRowIDs.Count)
            {
                oAcctFields = new ArrayList();

```

```

oAccountBC.ClearToQuery();
oAccountBC.SetViewMode(3);
oAccountBC.SetSearchSpec("Id", oAcctRowIDs(iRecordCount));
iRet = oAccountBC.ExecuteQuery(1);
iRet = oAccountBC.FirstRecord();
oAcctFields.Add(oAccountBC.GetFieldValue("Name"));
oAcctFields.Add(oAccountBC.GetFieldValue("City"));
oAcctFields.Add(oAccountBC.GetFieldValue("State"));
oAcctFields.Add(oAccountBC.GetFieldValue("Main Phone Number"));

oOrderBC.ClearToQuery();
oOrderBC.SetViewMode(3);
//oOrderBC.SetSearchExpr("[Account Id] = \"\" +
oAcctRowIDs(iRecordCount) + "\" AND [Status] = \"Shipped\"");
oOrderBC.SetSearchExpr("[Account Id] = \"\" + oAcctRowIDs(iRecordCount) + "\"");
oOrderBC.SetSortSpec("Created(DESCENDING)");
iRet = oOrderBC.ExecuteQuery(1);
iRet = oOrderBC.FirstRecord();
if (iRet != 0)
{
oAcctFields.Add(oOrderBC.GetFieldValue("Order Number"));
}
else
{
oAcctFields.Add(String.Empty);
}
oSRBC.ClearToQuery();
oSRBC.SetViewMode(3);
//oSRBC.SetSearchExpr("[Account Id] = \"\" +
oAcctRowIDs(iRecordCount) + "\" AND [Status] = \"Open\"");
oSRBC.SetSearchExpr("[Account Id] = \"\" + oAcctRowIDs(iRecordCount) + "\"");
oSRBC.SetSortSpec("Created(DESCENDING)");
iRet = oSRBC.ExecuteQuery(1);
iRet = oSRBC.FirstRecord();
if (iRet != 0)
{
oAcctFields.Add(oSRBC.GetFieldValue("SR Number"));
}

```

```

        }
        else
        {
            oAcctFields.Add(String.Empty);
        }
        oAcctRecords.Add(oAcctRowIDs(iRecordCount), oAcctFields);
        iRecordCount++;
    }
    //After performing the queries, we have the data and are ready to
update the spreadsheet;
    //however, check to make sure we haven't received a request to stop
refreshing data for this spreadsheet
    for (var sKeyName in htServerContextItems.Keys)
    {
        if (sKeyName.StartsWith("STOP::"))
        {
            if (String(htServerContextItems[sRequestID]).ToUpper() ==
sRequestingFile.ToUpper)
            {
                bStopPinging = true;

oServerContextAccessor.RemoveServerContextItem(sRequestID);
                break;
            }
        }
    }

    //Make some assumptions about which cells should hold the data and
update the spreadsheet
    if (bStopPinging == false)
    {
        try
        {
            oWorkBook = GetObject(sRequestingFile);
            if (oWorkBook != null)
            {
                oSheet = oWorkBook.Sheets[1];
            }
        }
    }
}

```

```

        //oSheet.Cells.Activate();
        //Assume the data portion of the sheet starts at cell A2
        //If this cell is empty, then we're doing a new population
of the data

        bWorkSheetEmpty = false;
        if (String(oSheet.Cells[2, 1].Value).Trim().length == 0 ||
String(oSheet.Cells[2, 1].Value).Trim().ToLower() == "undefined")
        {
            bWorkSheetEmpty = true;
        }
        iRowCount = 2;
        while (iRowCount < oAcctRowIDs.Count + 2)
        {
            if (bWorkSheetEmpty == true)
                {
                    sAcctRowID = oAcctRowIDs(iRowCount - 2);
                    oSheet.Cells[iRowCount, 1].Value = sAcctRowID;
                }
            else
            {
                sAcctRowID = String(oSheet.Cells[iRowCount,
1].Value);
            }
            oAcctFields = ArrayList(oAcctRecords(sAcctRowID));
            iColCount = 2;
            while (iColCount < oAcctFields.Count + 2)
            {
                if (String(oSheet.Cells[iRowCount, iColCount].Value)
!= oAcctFields(iColCount - 2))
                {
                    oSheet.Cells[iRowCount, iColCount].Value =
oAcctFields(iColCount - 2);
                }
                iColCount++;
            }
            iRowCount++;
        }

```

```

        }
    }
    catch(e)
    {
        //If the error is RPC_E_CALL_REJECTED, then the spreadsheet was
busy
        //Log the error, but otherwise keep running.
        //If it is another error, then re-throw the exception
        if (String(e).ToUpper().IndexOf("RPC_E_CALL_REJECTED") >= 0)
        {
            oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Warnings,
CTCommon.Constants.LogOpType.RoutineExecution, "Call to update spreadsheet data
returned RPC_E_CALL_REJECTED");
        }
        else
        {
            throw(e);
        }
    }
}
}

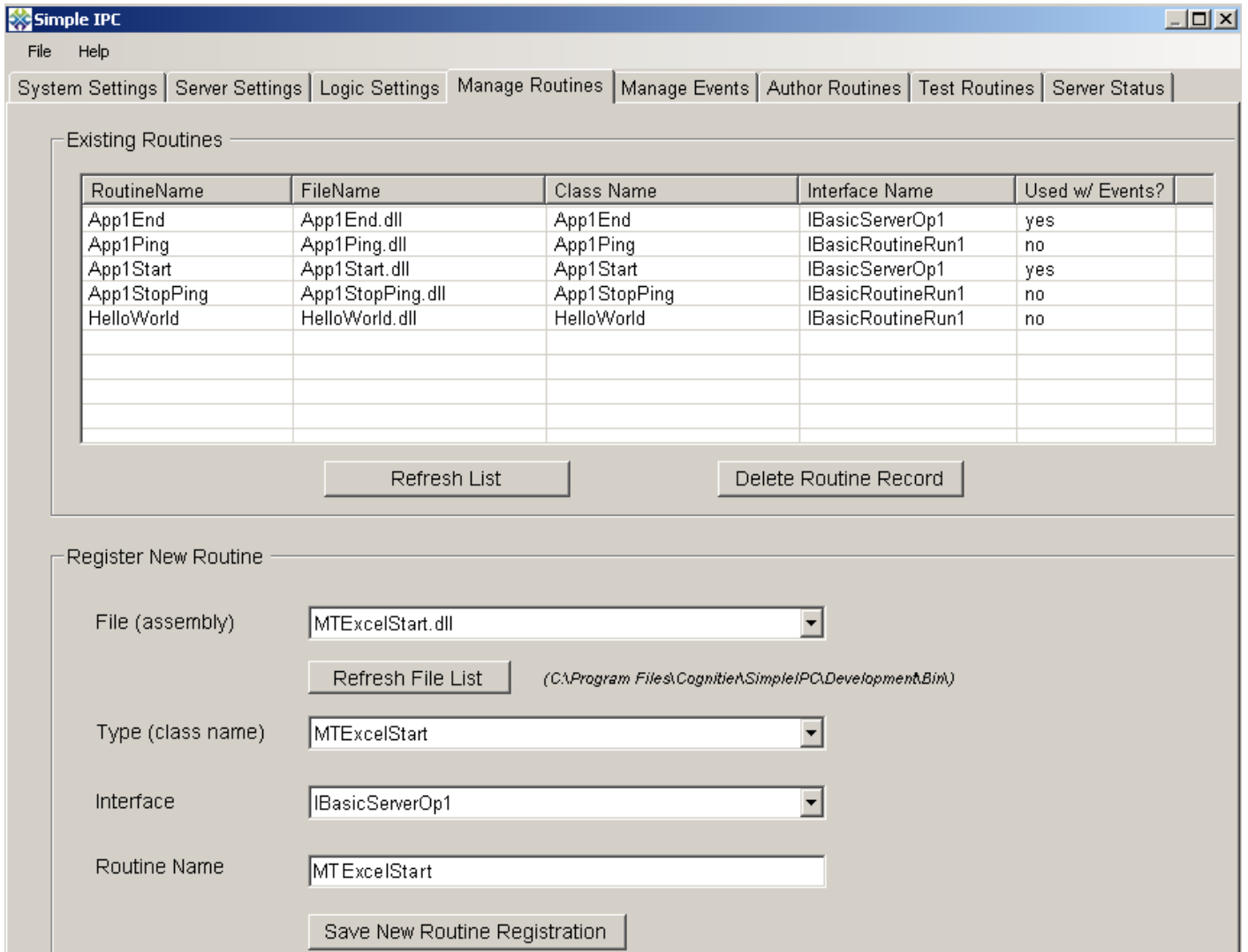
System.Threading.Thread.Sleep(1000);
htServerContextItems =
oServerContextAccessor.GetServerContextItems();
bServerShuttingDown =
System.Boolean(htServerContextItems["ServerShuttingDown"]);
}

//When the loop exits, perform cleanup operations - in production, do this
when an exception is thrown as well
oSheet = null;
oWorkBook = null;
oSRBC = null;
oSRBO = null;
oOrderBC = null;
oOrderBO = null;
oAccountBC = null;

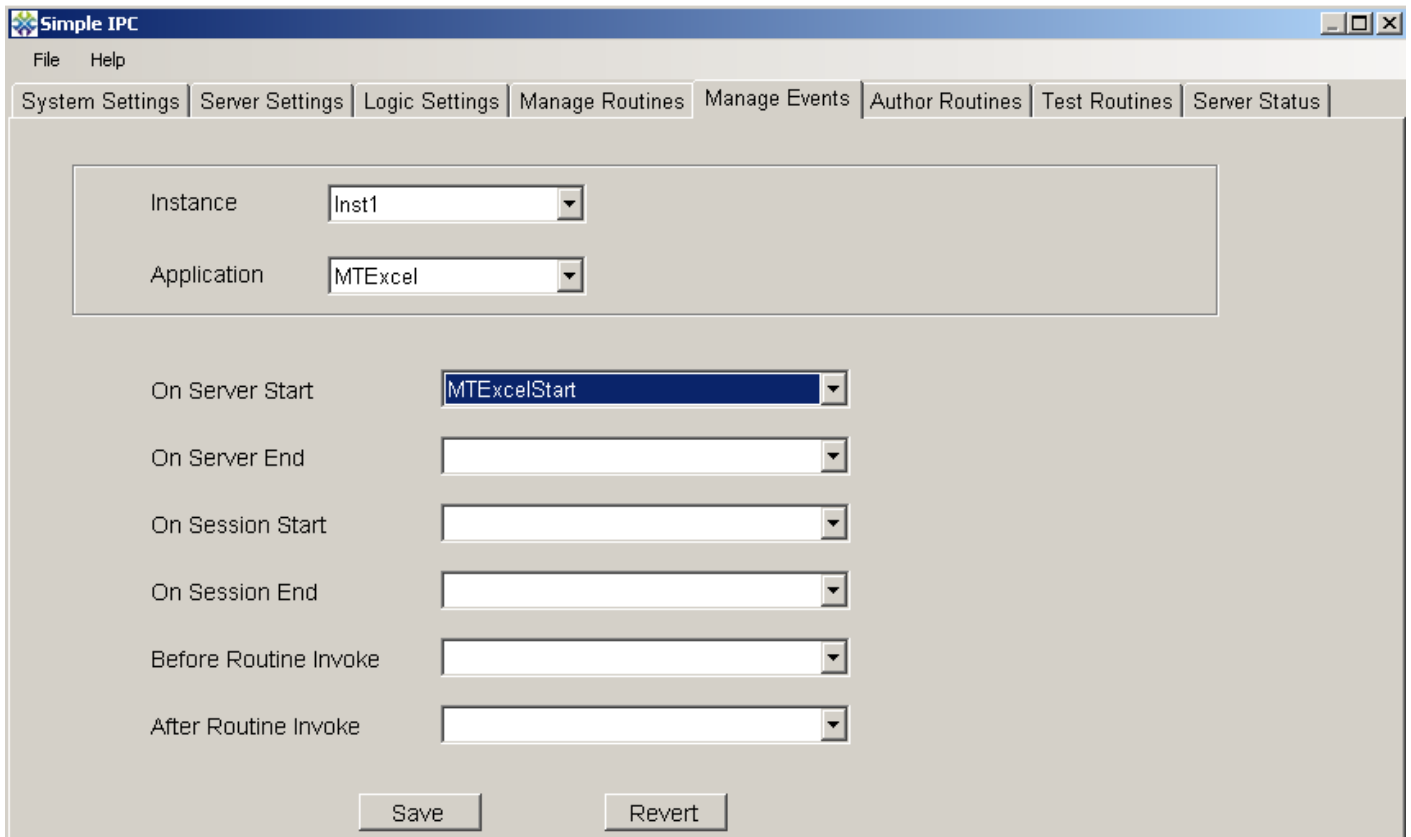
```

```
oAccountBO = null;
oSiebelApp.Logoff();
oSiebelApp = null;
}
catch (e)
{
    //Write errors to the server log file
    oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, "Error in ProcessInboundRequests
thread: " + e);
    //If we hit an error, terminate the server in order to allow the launch of
a replacement server
    System.Diagnostics.Process.GetCurrentProcess().Kill();
}
} //end function
} //end class
```

Step 5: Register the routine we just created. Go to the Manage Routines tab of the CTConsole and click the Refresh File List button under the “File (Assembly)” drop-down. Select the MTEXcelStart.dll dll, enter a name of MTEXcelStart and click Save New Routine Registration.



Step 6: Associate the server start-up routine with the “On Server Start” event for our application. Go to the Manage Events tab and toggle the Instance selection to refresh the list in the Applications drop-down. Select the MTExcel application and associate the MTExcelStart routine with the “On Server Start” event.



Step 7: Go to the Author Routines tab and create a new source code file with a name of MTEExcelPing. Use the BasicRoutineRun1 template. For this exercise, enter the following code to set a request into the server context to be picked up by the worker thread.

```
import System;
import System.Collections;
import System.Reflection;

[assembly: AssemblyVersion("1.0.0.0")]
public class MTEExcelping implements CTSHost.IBasicRoutineRun1
{
    public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor, oSessionContextAccessor:
CTSHost.SessionContextAccessor, oCallParamsAccessor:
CTSHost.CallParamsAccessor) : System.Boolean
    {
        var oLogUtil: CTCommon.LogUtil;
        var bRet: System.Boolean;
```

```

oLogUtil = CTCCommon.LogUtil.Instance;
bRet = false;
try
{
    if (oServerContextAccessor.ServerCallCount >=
(oServerContextAccessor.ServerCallCountLimit - 1))
    {
        //Set the boolean context variable to true to let the worker thread
know the server is exiting
        var bServerShuttingDown: System.Boolean = true;
        oServerContextAccessor.SetServerContextItem("ServerShuttingDown",
bServerShuttingDown);
        //Wait for two seconds to let the worker thread dispose of the COM
objects and then terminate the server
        System.Threading.Thread.Sleep(2000);
        System.Diagnostics.Process.GetCurrentProcess().Kill();
    }
    else
    {
        //Put a request to refresh the spreadsheet contents into the server
context
        if (oCallParamsAccessor.InputArgs.Count > 1)
        {
            //The arguments to this method from the VBA code in the spreadsheet
are the file name
            //and a pipe-delimited list of row ids
            //The request payload from this method to the worker thread is an
arraylist
            //The first element of the arraylist is the file name and the second
is an arraylist of row ids parsed from the incoming string
            var sRequestID: String = "REQ::" + oCallParamsAccessor.InputArgs[0]
+ "_" + System.Environment.TickCount;
            var sRowIDs: String = oCallParamsAccessor.InputArgs[1];
            var oRowIDs: String[] = sRowIDs.Split('|');
            var oIDsArrayList: ArrayList = new ArrayList();
            for (var sID in oRowIDs)
            {
                if (sID.length > 0)

```

```

        {
            oIDsArrayList.Add(sID);
        }
    }
    var oRequestPayload: ArrayList = new ArrayList();
    oRequestPayload.Add(oCallParamsAccessor.InputArgs[0]);
    oRequestPayload.Add(oIDsArrayList);
    oServerContextAccessor.SetServerContextItem(sRequestID,
oRequestPayload);
    }
}
bRet = true;
}
catch (e)
{
    //Set errors into errors collection - must be strings
    oCallParamsAccessor.OutputErrors.Add(e.Message);
    //Also write errors to the server log file
    oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
}
return bRet;
} //end function
} //end class

```

Step 8: Register the routine we just created. Go to the Manage Routines tab of the CTConsole and click the Refresh File List button under the “File (Assembly)” drop-down. Select the MTExcelPing.dll dll, enter a name of MTExcelPing and click Save New Routine Registration.

Step 9: Go to the Author Routines tab and create a new source code file with a name of MTExcelStopPing. Use the BasicRoutineRun1 template. For this exercise, enter the following code to set a request into the server context to be picked up by the worker thread which will discontinue polling to refresh the data in the spreadsheet.

```

import System;
import System.Collections;
import System.Reflection;

[assembly:AssemblyVersion("1.0.0.0")]
public class MTExcelStopPing implements CTSHost.IBasicRoutineRun1

```

```

{
    public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor, oSessionContextAccessor:
CTSHost.SessionContextAccessor, oCallParamsAccessor:
CTSHost.CallParamsAccessor) : System.Boolean
    {
        var oLogUtil: CTCommon.LogUtil;
        var bRet: System.Boolean;

        oLogUtil = CTCommon.LogUtil.Instance;
        bRet = false;
        try
        {
            if (oServerContextAccessor.ServerCallCount >=
(oServerContextAccessor.ServerCallCountLimit - 1))
            {
                //Set the boolean context variable to true to let the worker thread
know the server is exiting
                var bServerShuttingDown: System.Boolean = true;
                oServerContextAccessor.SetServerContextItem("ServerShuttingDown",
bServerShuttingDown);
                //Wait for two seconds to let the worker thread dispose of the COM
objects and then terminate the server
                System.Threading.Thread.Sleep(2000);
                System.Diagnostics.Process.GetCurrentProcess().Kill();
            }
            else
            {
                //Put a request to refresh the spreadsheet contents into the server
context
                if (oCallParamsAccessor.InputArgs.Count > 0)
                {
                    //The argument to this method from the VBA code in the spreadsheet
is the file name
                    var sRequestID: String = "STOP::" + oCallParamsAccessor.InputArgs[0]
+ "_" + System.Environment.TickCount;
                    oServerContextAccessor.SetServerContextItem(sRequestID,
oCallParamsAccessor.InputArgs[0]);
                }
            }
        }
    }
}

```

```

        }
    }
    bRet = true;
}
catch (e)
{
    //Set errors into errors collection - must be strings
    oCallParamsAccessor.OutputErrors.Add(e.Message);
    //Also write errors to the server log file
    oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
}
return bRet;
} //end function
} //end class

```

Step 10: Register the routine we just created. Go to the Manage Routines tab of the CTConsole and click the Refresh File List button under the “File (Assembly)” drop-down. Select the MTExcelStopPing.dll dll, enter a name of MTExcelStopPing and click Save New Routine Registration.

Step 11: Go to the Author Routines tab and create the routine that will execute when an IPC server exits. Create a new routine called MTExcelEnd using the BasicServerOp1 template. Enter the following code to set a Boolean variable in the server context which indicates to the worker thread that it should log off from Siebel and exit.

```

import System;
import System.Collections;
import System.Reflection;

[assembly:AssemblyVersion("1.0.0.0")]
public class MTExcelEnd implements CTSHost.IBasicServerOp1
{
    public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor) : System.Boolean
    {
        var oLogUtil: CTCommon.LogUtil;
        var bRet: System.Boolean;

        oLogUtil = CTCommon.LogUtil.Instance;
        bRet = false;
    }
}

```

```

    try
    {
        //Set the boolean context variable to true to let the worker thread know
the server is exiting
        var bServerShuttingDown: System.Boolean = true;
        oServerContextAccessor.SetServerContextItem("ServerShuttingDown",
bServerShuttingDown);

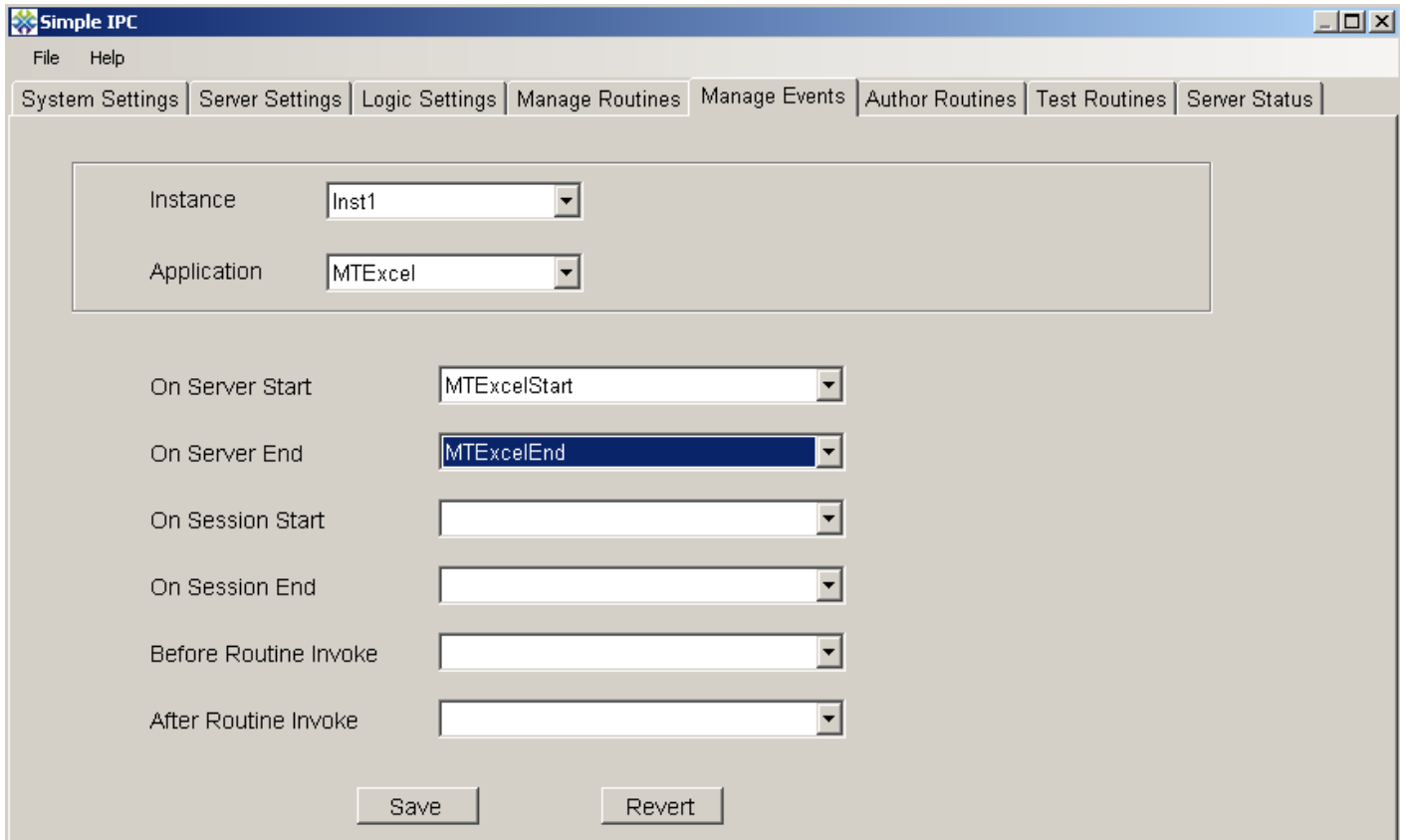
        //Wait for a minute to let the worker thread finish what it is doing and
then exit
        System.Threading.Thread.Sleep(60000);

        bRet = true;
    }
    catch (e)
    {
        //Write errors to the server log file
        oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
    }
    return bRet;
} //end function
} //end class

```

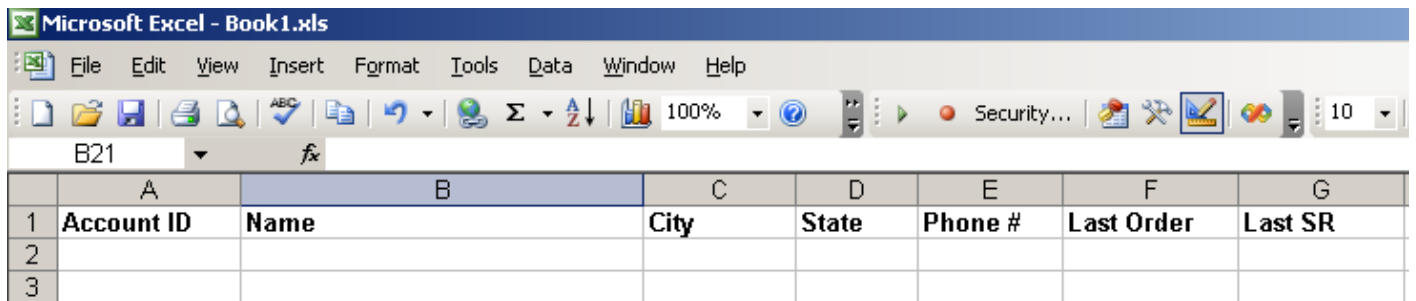
Step 12: Register the routine we just created. Go to the Manage Routines tab of the CTConsole and click the Refresh File List button under the “File (Assembly)” drop-down. Select the MTExcelEnd.dll dll, enter a name of MTExcelEnd and click Save New Routine Registration.

Step 13: Associate the server exit routine with the “On Server End” event for our application. Go to the Manage Events tab and toggle the Application selection to “App1” and then MTExcel to refresh the selections in the routine drop-downs. Associate the MTExcelEnd routine with the “On Server End” event of the MTExcel application.

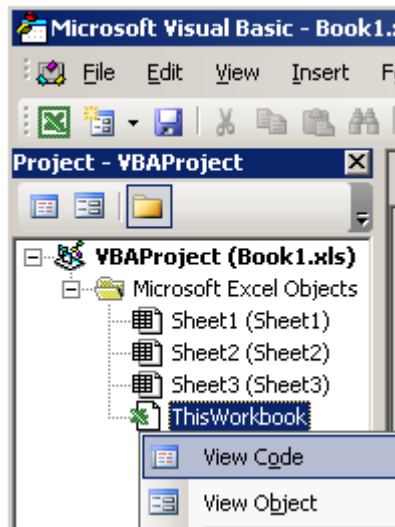


Step 14: Create a new MS Excel file (MS Excel 2003 is used in this example). Enter column header values in the first row of the first Sheet as follows:

- Account ID
- Name
- City
- State
- Phone #
- Last Order
- Last SR



Step 15: Go to the Visual Basic Editor, right-click on “This Workbook” and select “View Code”.



Step 16: Enter the following VBA code to interact with SimpleIPC and keep the spreadsheet contents up-to-date from the Siebel data source.

```
Public dSchedStart As Double

Private Sub Workbook_Open()
    Call DoPing
End Sub

Private Sub Workbook_BeforeClose(bCancel As Boolean)
On Error Resume Next
    'Stop Timer
    Application.OnTime EarliestTime:=dSchedStart,
Procedure:="ThisWorkbook.DoPing", Schedule:=False

    'Declare variables
    Dim oComp
    Dim sSessionID, sAppID, sRoutine, bRet
    Dim oInputArray, oOutputArray, oErrorArray, oWarningArray

    'Initialize variables
    sAppID = "MTEExcel"
    sRoutine = " MTEExcelStopPing"
    sSessionID = ""
```

```

'Initialize input arguments array - make the file name the first input
ReDim oInputArray(0)
oInputArray(0) = ThisWorkbook.Path & "\" & ThisWorkbook.Name

'Get reference to COM object and then call BasicRoutineCall1
Set oComp = CreateObject("CTOutProcCOMInst1.CTSessionComp1")
bRet = oComp.BasicRoutineCall1(sSessionID, sAppID, sRoutine, oInputArray,
oOutputArray, oErrorArray, oWarningArray)

'De-allocate
Set oInputArray = Nothing
Set oOutputArray = Nothing
Set oErrorArray = Nothing
Set oWarningArray = Nothing
Set oComp = Nothing
End Sub

Sub DoPing()
'Declare variables
Dim oComp
Dim sSessionID, sAppID, sRoutine, bRet
Dim oInputArray, oOutputArray, oErrorArray, oWarningArray
Dim sRowIDs, iRowCount, iLoopCount, iLoopMax

'Initialize variables
sAppID = " MTEExcel"
sRoutine = " MTEExcelPing"
sSessionID = ""

'Initialize input arguments array - make the file name the first input
ReDim oInputArray(1)
oInputArray(0) = ThisWorkbook.Path & "\" & ThisWorkbook.Name

'Collect the Row IDs (if any) from the spreadsheet and make them the second
input
iRowCount = 2

```

```

iLoopCount = 0
iLoopMax = 10
sRowIDs = ""
Do While iLoopCount < iLoopMax
    If CStr(ThisWorkbook.Sheets(1).Cells(2, 1)) = "" Then
        Exit Do
    Else
        If iLoopCount > 0 Then
            sRowIDs = sRowIDs & "|"
        End If
        sRowIDs = sRowIDs & CStr(ThisWorkbook.Sheets(1).Cells(2, 1))
    End If
    iLoopCount = iLoopCount + 1
Loop
oInputArray(1) = sRowIDs

'Get reference to COM object and then call BasicRoutineCall1
Set oComp = CreateObject("CTOutProcCOMInst1.CTSessionComp1")
bRet = oComp.BasicRoutineCall1(sSessionID, sAppID, sRoutine, oInputArray,
oOutputArray, oErrorArray, oWarningArray)

'De-allocate
Set oInputArray = Nothing
Set oOutputArray = Nothing
Set oErrorArray = Nothing
Set oWarningArray = Nothing
Set oComp = Nothing

'Set the next timer
dSchedStart = Now + TimeValue("00:03:00")
Application.OnTime EarliestTime:=dSchedStart,
Procedure:="ThisWorkbook.DoPing", Schedule:=True
End Sub

```

Step 17: Test your work. Close and re-open the MS Excel file. If you receive a security warning, then allow the “macros” to run. The start-up process will be time-consuming because a new IPC server is being launched and because the spreadsheet is initially empty. Perform the following validations:

- After the initial spreadsheet population, check the data against the actual data in Siebel for accuracy.
- Change some of the spreadsheet cell values (anything but the row id) to invalid values and wait 3-5 minutes. The values should be set back to the actual values from Siebel.
- Update some values in Siebel and wait 3-5 minutes. The new Siebel values should be reflected in the spreadsheet.
- After the initial population, the MS Excel application should be responsive to user input.
- Use Task Manager to terminate the CTRServerInst1.exe process, but leave MS Excel running. Within a few minutes, a new instance of CTRServerInst1.exe should be launched and the spreadsheet should be updated.
- Periodically check Task Manager for memory and CPU utilization of Excel.exe. Both should remain low. CPU utilization of CTRServerInst1.exe may spike up at times, but it should come back down.